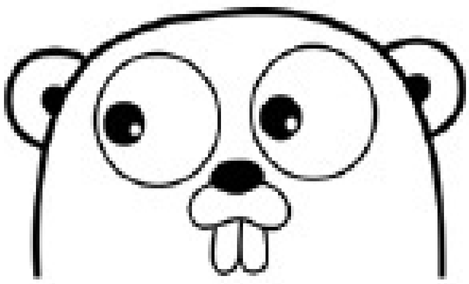


The Little Go Book

by Karl Seguin

目錄

《Go简易教程》

《The Little Go Book》中文译本，中文正式名《Go简易教程》。

目录

翻译进度

结论

致谢

- 本书原作者：Karl Seguin
- 参与翻译人员：
    - Jell

支持本书

如果你喜欢本书《Go简易教程》，联系lisong1205@gmail.com，可以参与到本书的翻译或纠正工作中来，欢迎PR。

你也可以关注我的微信公众号：reborncodinglife，欢迎讨论交流！

授权许可

The Little Go Book

by Karl Seguin

关于本书

授权许可

最新版本

这本书的最新版本可以在以下网站获得:

http://github.com/karlseguin/the-little-go-book

链接

- 目录
- 下一章：引言

引言

当开始学习一门新的语言时，我总有一种既爱又恨的感觉。一方面，语言是我们做事情的基础，即使是小的变化，也会带来明显的效果。当一些事情恍然大悟时，会给如何编程带来持久的影响，并且能够重新定义你关于其他语言的期望。不利的一面是，语言设计是一个持续的过程。学习新的关键字、系统类型、编码风格、新的库、社区和范例，似乎很难解释这些需要付出许多努力。和那些我们必须学习的事相比较，花费时间去学习新语言常让人觉得不值得。

也就是说，我们必须愿意采用渐进的步骤，我们必须进步。因为语言是我们做事的基础。虽然语言一直在变化，但是它们趋向于一个更广阔的范围，并且会影响生产力、可读性、可靠性、性能、可测试性、依赖管理、错误处理、文档、性能分析、社区和标准库等等。难道说千刀万剐导致的死亡是一种积极方式？

这也给我们留下了一个重要的问题，为什么是go语言？对我来说，有两个令人信服的原因。第一，这是个相对简单的语言，它有一个相对简单的标准库。在很多方面，go语言渐进的性质，将简化一些我们过去几十年所看到的增加到语言上的复杂性。第二，对于大多数开发者，这将会补充你现有的语言工具库。

go被创建成一种系统语言（比如，操作系统和设备驱动），go是针对C/C++开发者的。据go核心开发组说，我可以确定是真的，应用程序开发者已经成为主要的go语言用户，而不是系统开发者。为什么呢？我不能代表所有的系统开发人员，但是，对于构建网站、服务和桌面应用等而言，主要归结于一类新兴系统的需求，这类系统介于低级系统应用和高级系统应用之间。

可能go语言有消息传递机制、带缓存、重计算数据分析、命令行接口、日志或监控，我不知道给go语言什么样的标签，但是在我的职业生涯中，由于系统持续增长的复杂性和成千上万种常用的并发方式，很明显，定制基础设施系统，是一个不断增长的需求。你可以用ruby或python或别的语言建立这样的系统（好多人这么做），但是这类系统受益于更严格类型系统和更高性能。同样地，你可以用go语言构建网站(也有好多人这么做)，但是话又说回来，我还是喜欢通过表达性更强的Node或Ruby来实现这样的系统。

go语言还擅长于其他的领域。比如，当运行一个编译过的go程序时，它没有依赖性。你不必担心用户是否安装了ruby或者jvm，而且如果是这样，还要考虑是什么版本。出于这个原因，go作为命令行界面程序和其他并发类型应用程序的开发语言（例如日志收集），变得越来越流行。

坦白地说，学习go语言可以有效的利用你的时间。你不必花大量的时间去学习或者掌握它，你从你的努力中最终会得到一些实用的东西。

作者注解

我犹豫地写下这本书，有两个原因。首先，是由于go语言官方文档已经很完善了，特别是《Effective Go》。另一个原因是我在写一本介绍语言类的书时有点不安。当我写《The Little MongoDB Book》这本书时，我已经假设大多数读者理解关系型数据库和建模的基本知识。写《The Little Redis Book》时，你也可以做出类似的假设，即读者已经可以往redis中插入键值，然后从redis中查询该键值。

据我说知，前面这些章节，我不能再做出相同的假设。你花多长时间学习接口并理解它，这是个新的概念。比起go语言拥有的接口，其他人家是否需要更多？最终，如果你告诉我本书那些地方太浅或者太详细，考虑到这本书的价值，我会感到欣慰。

链接

- 目录
- 上一章：关于本书
- 下一章：准备工作

准备工作

链接

- 目录
- 上一章：引言
- 下一章：第1章

第1章 基础知识

go是一种编译型、具有静态类型和类c风格语法的语言，并具备垃圾回收机制，这意味着什么呢？

链接

- 目录
- 上一章：准备工作
- 下一节：编译

1.1 编译

编译是一个将源代码翻译成更低级语言的过程，例如汇编语言（例如go就是如此），或者其他中间语言（例如java和c#）。

编译型语言在使用过程中会有点不爽，因为编译比较慢。如果编译不得不花掉几十分钟或者几个小时，在快速迭代开发时往往是难以实现的。在设计go语言时，编译速度是主要的设计目标之一。对于大项目开发人员来说，这里有一个好消息，通过解释型语言，我们常能获得快速的反馈周期。

编译型语言倾向于运行得更快且在运行时没有额外的依赖关系（至少对于c、c++和go语言来说，直接编译成汇编语言是可行的）。

链接

- 目录
- 上一节：基础知识
- 下一节：静态类型

1.2 静态类型

静态类型语言意味着变量必须指定一个类型，例如整型、字符串、布尔型和数组等。可以在声明变量时指定变量类型，大多数情况下，让编译器自动去推断变量类型（我们将看到一些简单的例子）。

关于静态类型，有许多相关内容可以介绍，但是我相信，要理解静态类型最好的方法就是去阅读代码。如果你使用过动态类型语言，你可能会觉得静态类型有点繁琐。你的想法没错，但是静态类型也有很多优点，尤其编译静态类型的语言时。这2个观点经常被混为一谈。这是一个事实，但这不是一个硬性规定，你可以拥有同时动态类型和静态类型的语言。在一些严格类型的系统中，编译器只能检测程序的一些语法错误和更近一步的优化程序。

链接

- 目录
- 上一节：编译
- 下一节：类c语法

1.3 类c语法

一般说一门语言具有类c语法意味着如果你习惯使用其他类c语言例如c、c++、java、javascript和c#，然后你就会发现go语言也是类似，至少表面上是。例如，使用 `&&` 表示一个布尔运算 `AND` , `==` 用于相等比较， `{` 和 `}` 表示一个代码段的开始和结束，并且数组的索引值是从0开始。

类c语法也意味着一行代码以分号结尾，条件语句使用圆括号。go语言不需要这2种方式，尽管圆括号依然用于控制优先级。例如，一个 `if` 语句是这样的：

```
if name == "Leto" {
    print("the spice must flow")
}
```

在更复杂的情况下，圆括号依然很有用：

```
if (name == "Goku" && power > 9000) || (name == "gohan" && power < 4000) {
    print("super Saiyan")
}
```

除此之外，go比c#或者java更接近c，不但在语法上类似，还有一定的目的。这反映在语言风格的简单和整洁上，随着不断深入学习，你会越来越明显的体会到这种特性。

链接

- 目录
- 上一节：静态类型
- 下一节：垃圾回收

1.4 垃圾回收

当创建一些变量时，变量有一个确定的生命周期。例如函数中定义的局部变量，当函数退出时变量就不存在了。另外在其他情况下，至少对于编译器来说，这不是那么的明显。例如，某个被函数返回的变量的生命周期，或者被其他变量和对象引用的变量的声明周期，都是很难去判断的。如果没有垃圾回收机制，开发人员需要一直释放一些不再需要的变量的内存。怎么实现？例如在c中，你需要正确的去释放一个变量的内存 `free(str)` 。

语言的垃圾回收机制（例如：ruby、python、java、javascript、c#和go）可以记录这些不再使用的变量，然后释放他们占用的内存。垃圾回收机制会带来一些性能影响，但是它也能消除很多毁灭性的bug。

链接

- 目录
- 上一节：类c语法
- 下一节：运行go代码

1.5 运行go代码

现在，让我们通过创建一段简单的程序开启我们的go学习旅程，并学习如何编译和执行一个程序。打开你最喜欢个一个文本编辑器，键入以下代码：

```
package main

func main() {
    println("it's over 9000!")
}
```

将文件保存为 `main.go` ，然后，将文件保存在任意位置，对于简单的例子，我们不需要深入到go的工作空间中。

接下来，打开一个命令行终端，进入 `main.go` 文件所在的目录。对我来说，我是输入了 `cd ~/code` 。

最后，键入以下命令运行这段程序：

```
go run main.go
```

如果程序正常运行，你应该会看到输出 `it's over 9000!` 。

但这是如何实现编译呢？为了方便起见， `go run` 会先编译然后再运行你的代码，它会在一个临时的目录下编译这段代码，然后执行，最后自动清除生成的临时文件。通过运行以下命令你可以看见这个零临时文件的位置：

```
go run --work main.go
```

如果你只想编译代码，使用 `go build` :

```
go build main.go
```

这会生成一个可执行文件 `main` ，你可以直接运行它。在linux/osx中，不要忘记在可执行文件前面加上点和反斜杠，所以你需要输入 `./main` 运行程序。

1.5.1 main

希望我们刚刚执行的代码可以被理解，我们定义了一个函数，并调用了内置函数 `println` 输出一个字符串。难道因为这里仅有一个选择，所以 `go run` 知道执行什么吗？不是的，在go语言中，程序的入口是 `main` 包中的 `main` 函数。

我们在后面的章节中会专门介绍包的相关内容，现在，我们暂时专注于理解go语言的基础知识，所以我们一直在 `main` 包中编写代码。

如果你愿意，你也可以改变代码并改变包的名字，并使用`go run`去执行，你会得到一个错误信息。然后，将包名改成`main`，但是函数名不叫`main`，再次运行代码，你会得到一个不同的错误信息。使用`go build`进行相同的操作，注意编译代码时，这里没有运行代码的入口点。这是很正常的，例如当你编译一个库时。

链接

- 目录
- 上一节：垃圾回收
- 下一节：导入包

1.6 导入包

go有很多内置的函数，例如 `println` ，不需要引用即可使用。但是如果不借助go的标准库或者第三方库，我们能做的事情有限。在go中，使用关键字 `import` 在代码中导入一个包并使用。

修改我们的程序：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    if len(os.Args) != 2 {
        os.Exit(1)
    }
    fmt.Println("It's over ", os.Args[1])
}
```

使用下面的命令运行：

`go run main.go 9000`

我们现在使用了2个go的标准包： `fmt` 和 `os` 。我们也引入了另外一个内置函数 `len` 。 `len` 返回一个字符串大小或者一个字典中值的个数，或者如上代码所示，返回数组元素的个数。如果你想知道为什么这里我们使用2个参数，因为第一个参数即索引为 `0` 一直表示当前正在运行的可执行文件的路径（你可以自己修改程序并打印观察）。

你可能已经注意到了我们在函数名前加了包名作为前缀，例如， `fmt.Println` 。这和其他许多语言不同。我们将会在接下来的章节学习更多关于包的内容。现在，只需知道怎么导入并使用包就是一个很好的开端。

go在导入包的时候是比较严格的，如果导入的包没有被使用，那么程序不能被编译。试着运行一下代码：

```
package main

import (
    "fmt"
    "os"
)

func main() {
}
```

你会得到2个错误信息，提示 `fmt` 和 `os` 包被导入但是没有被使用。你会觉得很不适应么？但是，过一段时间，你会变得适应（虽然任然很烦人）。go之所以这么严格是因为如果没有使用导入的包会使编译变慢。不可否认，这个问题我们很多人都没有考虑到。

另外，需要值得注意的是go的标准库提供了非常详细的文档。你可以在http://golang.org/pkg/fmt/#Println查询到更多关于 `Println` 函数的信息。你甚至可以点击章节标题查看源码。你也可以滚动到顶部学习更多关于go格式化输出的功能。

如果你不能上网，你可以在本地运行下面的命令获取这个文档：

`godoc -http=:6060`

并打开浏览器，输入 `http://localhost:6060` 。

链接

- 目录
- 上一节：运行go代码
- 下一节：变量和声明

1.7 变量和声明

这将是美好的开始和结束，通过写下 `x = 4` ，我们查看变量，可以说声明了一个变量并赋值，但是很不幸，go语言变量声明和赋值比这更复杂。通过学习一些简单的示例开始学习变量声明和赋值。然后在下一章，当我们创建并使用结构体时，我们会深入学习。尽管如此，你需要花一些时间去适应。

你可能会惊讶，为什么会如此复杂，让我们以一些例子开始学习。

在go中最直接的方式去声明变量并赋值也是最繁琐的：

```
package main

import (
    "fmt"
)
func main() {
    var power int
    power = 9000
    fmt.Printf("It's over %d\n", power)
}
```

在这里我们声明了一个 `int` 型变量 `power` 。默认情况下，go会给 `power` 赋一个 `0` 值。整型赋 `0` ，布尔型赋 `false` ，字符串型赋 `""` 等。然后，我们将 `9000` 赋值给变量 `power` 。我们也可以将2行代码合并成一行：

`var power int = 9000`

虽然需要输入很多。go有一种方便简洁的变量声明操作符： `:=` ，go可以推断变量的类型：

`power := 9000`

这里有一个简洁的写法，通过函数也能正常工作：

```
func main() {
    power := getPower()
}
func getPower() int {
    return 9001
}
```

这里需要谨记， `:=` 用于声明一个变量并给变量赋值。为什么会这样？因为一个变量不能被声明2次（不在相同的代码范围）。如果你试着运行下面代码，你将会得到一个错误信息。

```
func main() {
    power := 9000
    fmt.Printf("It's over %d\n", power)
    // COMPILER ERROR:
    // no new variables on left side of :=
    power := 9001
    fmt.Printf("It's also over %d\n", power)
}
```

编译器会提示 `:=` 左边不是一个新变量。这意味着当我们第一次声明变量时，我们使用 `:=` 。但是在随后的赋值，我们要使用 `=` 。这有很多意义，但这也随时提醒着你何时该使用 `:=` 和 `=` 。

如果你仔细阅读错误信息，你将发现有多个变量。因为go支持多个变量同时赋值（使用 `=` 或者 `:=` ）：

```
func main() {
    name, power := "Goku", 9000
    fmt.Printf("%s's power is over %d\n", name, power)
}
```

另外，如果一个变量是新变量也可以使用 `:=` 进行赋值。例如：

```
func main() {
    power := 1000
    fmt.Printf("default power is %d\n", power)

    name, power := "Goku", 9000
    fmt.Printf("%s's power is over %d\n", name, power)
}
```

尽管变量 `power` 使用了 `:=` ，但是编译器不会在第2次使用 `:=` 时报错，因为这里有一个变量 `name` ，这是一个新的变量，允许使用 `:=` 。但是你不能改变 `power` 的类型。它已经被声明成一个整型，只能赋值整数。

现在，需要知道的最后一件事是，类似包导入，go程序中不能存在未使用的变量，例如：

```
func main() {
    name, power := "Goku", 1000
    fmt.Printf("default power is %d\n", power)
}
```

这段代码不能编译，因为变量 `name` 已经声明，但是没有被使用。类似未使用的导入包，可能这会让你有点失望，但是总的来看，我认为这是为了让代码更加的简洁和具有可读性。

接下来不在介绍关于变量的声明和赋值相关内容了。现在，你只需要记住使用 `var NAME TYPE` 声明一个变量，并且变量的初始值为它相应类型的零值，使用 `NAME := VALUE` 声明一个变量并赋值，使用 `NAME = VALUE` 去给已经声明过的变量赋值。

链接

- 目录
- 上一节：导入包
- 下一节：函数声明

1.8 函数声明

这是一个很好的机会去介绍函数支持多值返回。查看下面3个函数：一个没有返回值，一个返回一个值，一个返回2个值。

```
func log(message string) {
}

func add(a int, b int) int {
}

func power(name string) (int, bool) {
}
```

我们常常这样使用最后一种函数：

```
value, exists := power("goku")
if exists == false {
    // 处理出错情况
}
```

如果你只想获得返回值中的某个值，这种情况下，你可以将另外一个返回值赋给 `_`：

```
_, exists := power("goku")
    if exists == false {
    // 处理出错情况
}
```

这不仅仅是一种约定。`_` 是一个空白标识符，尤其在用在返回值时它没有真正的赋值。你可以一直使用 `_`，无论返回值是什么类型。

最后，你可能遇到一些不同的函数声明方式，即如果函数的参数都是相同的类型，那么可以使用以下的简洁方式定义：

```
func add(a, b int) int {

}
```

你会常常遇见函数返回多个值。你也会经常使用 `_` 去丢弃一个值。具名返回值和不详细的参数声明并不常见。但是迟早你都会遇到，所以了解他们是很重要的。

链接

- 目录
- 上一节：函数声明

- 下一节：继续之前

1.9 继续之前

我们已经学习了许多的小知识点，你可能会觉得有点脱节。我们有希望逐步构建一个很大的例子，然后将这些小知识点都使用上。

如果你是一个动态类型语言使用者，你可能会觉得go的变量类型和声明似乎更加复杂了。我同意你的看法。对于一些系统，动态类型的语言绝对更有效率。

如果你是一个静态类型语言使用者，你可能会习惯使用go。类型推断和多值返回是如此的美好（尽管这不是go独有的）。希望伴随着我们的不断深入学习，你会喜欢上go干净和简洁的语法规则。

链接

- 目录
- 上一节：函数声明
- 下一章：结构体

2.0 结构体

go不是像c++、java、ruby和c#一样的面向对象语言。它没有对象和继承的概念。因此也没有很多面向对象语言的特性如多态和重载。

go提供了结构体，并且可以将一些方法和结构体关联。go也支持一种简单但是更有效的组合形式。总的来说，这是为了让代码更加简洁，但是在一些场合，你会失去一些面向对象语言提供的特性。（需要特别指出的是，通过组合实现继承是一个很古老的方式了，但是go是我使用过的所有语言中，立场最坚定的。）

尽管你不能像你之前使用的面向对象语言一样使用go，但是你将会注意到，定义一个结构体和定义一个类是很相似的。这里给出了一个简单的例子，定义一个 `Saiyan` 结构体：

```
type Saiyan struct {
    Name string
    Power int
}
```

我们很快会看到怎么往这个结构体添加一个方法，就像类会拥有方法一样。在这之前，我们需要去学习如何声明结构体。

链接

- 目录
- 上一章：继续之前
- 下一节：声明和初始化

2.1 声明和初始化

当我们第一次看见变量和声明时，我们仅仅看见一些内置的类型，比如整型和字符串。现在我们将学习结构体，并且我们会深入学习包括指针的内容。

通过一种最简单的方式去创建一个结构体值类型：

```
goku := Saiyan{
    Name: "Goku",
    Power: 9000,
}
```

注意：上面的结构体中，结尾的逗号 `,` 是不能省的。如果没有逗号，编译器会给出一个错误。你将喜欢上这种一致性要求，特别是如果你已经使用一种相反的语言或格式。

我们不需要给结构体设置任何值甚至任何字段。这2中方式都是有效的：

```
goku := Saiyan{}

// 或者

goku := Saiyan{Name: "Goku"}
goku.Power = 9000
```

这就像一个未赋值的变量一样，结构体的字段也会有一个0值。

另外，你也可以省略字段的名字，按字段的顺序进行声明（尽管为了简洁起见，你尽量在结构体只有少量字段时才使用这种方式）：

`goku := Saiyan{"Goku", 9000}`

上面的例子主要是声明了一个变量 `goku` ，并给它赋值。

尽管在大多数时候，我们不希望一个变量直接关联一个值，而是希望一个指针指向变量的值。指针是一个内存地址。通过指针可以找到这个变量实际的值。这是一种间接的取值。严格地说，这与存在一个房子并指向另外一个房子有一些区别。

为什么我们需要一个指针指向一个值，而不需要一个实际值。这主要是因为在go语言中，函数的参数传递都是按值传递，即传递的是一个拷贝。了解到这点，下面程序会打印什么？

```
func main() {
    goku := Saiyan{"Goku", 9000}
    Super(goku)
    fmt.Println(goku.Power)
}

func Super(s Saiyan) {
    s.Power += 10000
}
```

答案是9000，不是19000。为什么？因为 `Super` 只是改变了 `goku` 的一个拷贝，所以在 `Super` 中的改变不会调用者中反应出来。如果你希望答案是19000，我们需要传递一个指向我们值的指针：

```
func main() {
    goku := &Saiyan{"Goku", 9000}
    Super(goku)
    fmt.Println(goku.Power)
}
func Super(s *Saiyan) {
    s.Power += 10000
}
```

我们改变了2个地方。首先是使用了 `&` 操作符去获得我们值的地址（ `&` 叫取地址符）。接下来，我们改变了 `Super` 接受的参数类型。之前我们是传递一个 `Saiyan` 的值类型，现在我们传递了一个地址类型 `*Saiyan` ，这里的 `*X` 表示一个指向类型 `X` 的一个指针。显而易见， `Saiyan` 和 `*Saiyan` 类型之间有一定的联系，但是它们是两种不同的类型。

需要指出的是，我们现在传递给 `Super` 参数的仍然是 `goku` 的值拷贝。只是现在 `goku` 的值变成了一个地址。这个地址拷贝和源地址相同。可以认为它类似一个指向餐厅方向的拷贝，这就间接服务于我们。虽然是一个拷贝，但是和源地址一样，也指向同一个餐厅。

我们能证明这是一个拷贝，通过试着去改变它指向的地方（这可能不是你想做的）：

```
func main() {
    goku := &Saiyan{"Goku", 9000}
    Super(goku)
    fmt.Println(goku.Power)
}
func Super(s *Saiyan) {
    s = &Saiyan{"Gohan", 1000}
}
```

上面的代码在此输出了9000。很多语言也有类似的行为，包括ruby、python、java和c#。go某种程度上和c#一样，只是让事实可见。

显而易见，复制一个指针变量的开销比复制一个复杂的结构体小。在一个64的系统上，指针的大小只有64位。如果我们的结构体有很多字段，创建一个结构体的拷贝会有很大的性能开销。指针的真正意义就是通过指针可以共享值。我们想通过 `Super` 去改变 `goku` 的拷贝或者改变共享的 `goku` 值本身？

这里不是说你需要一直使用指针。本章的结尾，在我们学到更多关于结构体使用的内容之后。我们将重新审视值类型和指针类型的问题。

链接

- 目录
- 上一节：结构体

- 下一节：结构体上的函数

2.2 结构体上的函数

我们可以将一个方法和一个结构体关联：

```
type Saiyan struct {
    Name string
    Power int
}

func (s *Saiyan) Super() {
    s.Power += 10000
}
```

在上面的代码中，我们可以说类型 `*Saiyan` 是 `Super` 方法的接收者。可以向下面代码一样调用 `Super`：

```
goku := &Saiyan{"Goku", 9001}
goku.Super()
fmt.Println(goku.Power) // 将打印：19001
```

链接

- 目录
- 上一节：声明和初始化
- 下一节：构造函数

2.3 构造函数

结构体没有构造函数，你可以创建一个函数返回一个相应类型的实例代替（类似一个工厂）：

```
func NewSaiyan(name string, power int) *Saiyan {
    return &Saiyan{
        Name: name,
        Power: power,
    }
}
```

这种模式会导致开发者犯一些错误。另外，这有点轻微的语法变化；其次，让人觉得不好区分。

我们的工厂函数没有必要返回一个指针；下面代码是完全有效的：

```
func NewSaiyan(name string, power int) Saiyan {
    return Saiyan{
        Name: name,
        Power: power,
    }
}
```

链接

- 目录
- 上一节：结构体上的函数
- 下一节：new

2.4 new

尽管没有构造函数，go有一个内置的函数 `new` ，可以用来分配一个类型需要的内存。 `new(X)` 和 `&X{}` 是等效的：

```
goku := new(Saiyan)
// 等效
goku := &Saiyan{}
```

用那种方式取决于你，但是你会发现，当需要去初始化结构体字段时，大多数人更喜欢使用后者，因为后者更易读：

```
goku := new(Saiyan)
goku.name = "goku"
goku.power = 9001

//对比

goku := &Saiyan {
    name: "goku",
    power: 9000,
}
```

无论你选择那种方式，如果你选择上面的工厂模式，你可以隐藏一些代码细节，但是需要留意任何内存分配细节。

链接

- 目录
- 上一节：构造函数
- 下一节：结构体字段

2.5 结构体字段

在之前的例子中，我们已经知道 `Saiyan` 有2个字段 `Name` 和 `Power` ,类型分别是字符串型和整型。字段可以是任意的类型，包括其他结构体类型或者我们还没有接触过的类型如：数组、映射、接口和函数类型。

例如，我们可以扩展 `Saiyan` 的定义：

```
type Saiyan struct {
    Name string
    Power int
    Father *Saiyan
}
```

通过下面方式初始化：

```
gohan := &Saiyan{
    Name: "Gohan",
    Power: 1000,
    Father: &Saiyan {
        Name: "Goku",
        Power: 9001,
        Father: nil,
    },
}
```

链接

- 目录
- 上一节：new
- 下一节：组合

2.6 组合

go支持组合，即一种结构体包含另外一个结构体。在一些语言中，这叫混入类或者特性。语言总是不能实现简明的组合机制。在java中：

```
public class Person {
    private String name;

    public String getName() {
        return this.name;
    }
}


public class Saiyan {
    // 这表明`Saiyan`有一个`person`
    private Person person;
    // 可以使用`person`调用方法
    public String getName() {
        return this.person.getName();
    }
    ...
}
```

这样语法太繁琐了。每个 `Person` 的方法在 `Saiyan` 中都被复写一遍。go避免这样繁琐的方式：

```
type Person struct {
    Name string
}

func (p *Person) Introduce() {
    fmt.Printf("Hi, I'm %s\n", p.Name)
}

type Saiyan struct {
    *Person
    Power int
}

// 使用:
goku := &Saiyan{
    Person: &Person{"Goku"},
    Power: 9001,
}
goku.Introduce()
```

结构体 `Saiyan` 有一个字段时 `*Persion` 类型。因此我们没有明确的给它一个字段名，我们可以间接的使用这个组合类型的字段和方法。然而，go编译器给会给该字段一个名字，认为这是完全有效的。

```
goku := &Saiyan{
    Person: &Person{"Goku"},
}

fmt.Println(goku.Name)
fmt.Println(goku.Person.Name)
```

上面代码都将打印 `Goku` 。

组合优于继承吗？很多人都认为组合是一种更健壮的共享代码的方式。当你使用继承，你的类将和你的超类紧耦合，并且你最终更关注继承，而不是行为。

2.6.1 重载

虽然重载不是针对结构体，但是也值得提及。简单来说，go不支持重载。因此你会看见（和写）很多函数诸如 `Load` 、 `LoadById` 和 `LoadByName` 等等。

然而，因为匿名组合只是一个编译技巧，我们能“重写”一个组合类型的方法。例如，我们的结构体 `Saiyan` 可以定义自己的 `Introduce` 方法：

```
func (s *Saiyan) Introduce() {
    fmt.Printf("Hi, I'm %s. Ya!\n", s.Name)
}
```

这种组合版本总是可以通过 `s.Person.Introduce()` 调用 `Introduce()` 方法。

链接

- 目录
- 上一节：结构体字段
- 下一节：指针类型和值类型

2.7 指针类型和值类型

当你写go代码时，很自然的就会问自己，这里应该使用值类型还是指针类型？这有2则好消息。首先，尽管接下来我们要讨论，但是答案都是一样的。

- 一个局部变量赋值
- 结构体字段
- 函数返回值
- 传递给函数的参数
- 方法的接收者

其次，如果你不确定使用那个，那么就使用指针。

正如我们所见，传递值类型是一种确保数据不可变的好方法（在函数内的改变不会影响到调用的代码）。有些时候，这是你需要的行为，但是更多时候，这不是你想要的。

即使你不打算改变数据，也要考虑到创建一个大结构体拷贝的开销。相反地，你可能有一个小结构体，例如：

```
type Point struct {
    X int
    Y int
}
```

这种情况下，拷贝一个结构体的开销可能被直接访问 `X` 和 `Y` 抵消了，而不是通过间接访问。

另外，这些都是些很微妙的情况，除非你是遍历成千上万个指针，否则你不会发现有任何差别。

链接

- 目录
- 上一节：组合
- 下一节：继续之前

2.8 继续之前

从实用性的方面来看，这章介绍了结构体，了解如何创建一个结构体实例的方法并指定接收者，并添加了指针到我们已经学习过的go类型系统中。接下来的一章主要基于我们已经学过的结构体知识和其内部工作机制。

链接

- 目录
- 上一节：指针类型和值类型
- 下一章：映射、数组和切片

3.0 映射、数组和切片

到目前为止，我们已经见过许多简单的类型和结构体。现在是时候学习数组、切片和映射了。

链接

- 目录
- 上一章：继续之前
- 下一节：数组

3.1 数组

如果你使用过python、ruby、perl、javascript或者php，也许你已经在写代码时使用过动态数组，这些数组在添加数据时会动态改变自己的大小。和大多数语言一样，在go中，数组是固定大小的。声明一个数组时我们必须指定它的大小，一旦数组的大小被指定，它就不能扩展变大：

```
var scores [10]int
scores[0] = 339
```

上面定义的数组可以容纳10个元素，使用索引 `scores[0]` 到 `scores[9]` 。当你尝试着访问超出数组边界的的元素，会导致一个编译错误或者运行时错误。

我们可以直接使用值初始化一个数组：

`scores := [4]int{9001, 9333, 212, 33}`

也可以使用 `len` 得到数组的长度， `range` 也可以遍历一个数组：

```
for index, value := range scores {

}
```

数组效率高但是不灵活。我们提前处理数据时，一般都不知道元素的数量。因此，我们使用切片。

链接

- 目录
- 上一节：映射、数组和切片
- 下一节：切片

3.2 切片

在go中，你一般很少直接使用数组。相反，你会使用切片。切片是一个轻量级的结构体封装，这个结构体被封装后，代表一个数组的一部分。这里指出了一些创建切片的方式，并指出我们以后会在何时使用这些方式。

第一种方式和我们创建一个数组时有点细微的变化：

`scores := []int{1,4,293,4,9}`

和声明数组不同的是，声明切片不需要在方括号中指定其大小。理解2中不同的方式创建切片，下面使用另外一种方式创建切片，这里使用 `make` ：

`scores := make([]int, 10)`

我们使用 `make` ，没有使用 `new` ，是因为创建一个切片不仅仅是分配一段内存（ `new` 只能分配一段内存）。需要特别指出的是，我们需要为底层数组分配内存，并且也要初始化这个切片。在上面的例子中，我们初始化了一个长度和容量都是10的切片。长度表示切片的长度，容量表示底层数组的大小。在使用 `make` 创建切片时，我们可以分别的指定长度和容量大小：

`scores := make([]int, 0, 10)`

上面创建了一个长度为0但是容量为10的切片。（如果你留意过，你会发现 `make` 和 `len` 实现了重载。go语言的一些特性会让你有点失望，因为有些特性是没有暴露出来给开发者使用。）

要更好的理解切片长度和容量之间的相互作用，让我们看下面的例子：

```
func main() {
    scores := make([]int, 0, 10)
    scores[5] = 9033
    fmt.Println(scores)
}
```

上面的例子不能运行。为什么？因为我们创建的切片长度是0。是的，这个底层的数组有10个元素。但是为了访问切片元素，我们需要明确的扩展切片。一种扩展切片的方式是使用 `append` ：

```
func main() {
    scores := make([]int, 0, 10)
    scores = append(scores, 5)
    fmt.Println(scores) // 打印：[5]
}
```

但是这改变为了我们之前代码的意图。当往一个长度为0的切片上添加元素时，这个元素会被赋值给切片的第一个元素。不管出于什么原因，那段不能运行的代码想给切片的第6个元素赋值。为了到达这个目的，我们可以再切分一直我们的切片：

```
func main() {
    scores := make([]int, 0, 10)
    scores = scores[0:6]
    scores[5] = 9033
    fmt.Println(scores)
}
```

调整切片的大小上限是多少？这个上限就是切片的容量大小，在上面的例子中，上限是10。你也许会认为，这实际没有解决定长数组的问题。事实证明，`append` 比较特殊。如果底层的数组已经达到上限，`append` 会重新创建一个更大的数组，并将所有的值复制过去（这就是动态数组的工作原理，例如php、python、ruby和javascript等等）。这就是为什么我们在上面的例子中使用 `append` 。我们必须将 `append` 返回的值重新赋予给 `scores` 变量：如果原始切片没有更多的空间时，`append` 可能会创建一个新值。

如果我告诉你go扩展数组使用的是2倍算法（2x algorithm）。你能猜出下面代码将输出什么吗？

```
func main() {
    scores := make([]int, 0, 5)
    c := cap(scores)
    fmt.Println(c)
    for i := 0; i < 25; i++ {
        scores = append(scores, i)
        // 如果容量已经改变，go为了容下这些新数据，不得不增长数组的长度
        if cap(scores) != c {
        c = cap(scores)
        fmt.Println(c)
        }
    }
}
```

切片 `scores` 的初始容量是5。但是为了容纳20个元素，切片的容量必须扩展3次，分别是12、24和48（译者注：原文不是12、24和48，而是10、20和40）。

作为最后一个例子，思考一下：

```
func main() {
    scores := make([]int, 5)
    scores = append(scores, 9332)
    fmt.Println(scores)
}
```

当面的代码输出是 `[0, 0, 0, 0, 0, 9332]` 。也许你认为应该是 `[9332, 0, 0, 0, 0]` 。对人类来说，这似乎更合乎逻辑。但是对于编译器来说，你是要告诉它往一个已经拥有5个元素的切片添加一个值。

最后，这里提供了4种常用的方式去初始化一个切片：

```
names := []string{"leto", "jessica", "paul"}
checks := make([]bool, 10)
var names []string
scores := make([]int, 0, 20)
```

你该使用哪一个？第一种你不需要太多的说明。但是使用这种方式你得提前知道你想往数组存放的值。

第二种方式在你想往切片的特定位置写入一个值时很有用，例如：

```
func extractPowers(saiyans []*Saiyans) []int {
    powers := make([]int, len(saiyans))
    for index, saiyan := range saiyans {
    powers[index] = saiyan.Power
    }
    return powers
}
```

第三种方式会返回一个空切片，一般和 `append` 一起使用，此时切片的元素数量是未知的。

最后一种方式可以让我们指定切片的初始容量。当我们大概知道需要多少元素时很有用。即使你知道元素的个数， `append` 也能被使用，这主要取决于个人喜好：

```
func extractPowers(saiyans []*Saiyans) []int {
    powers := make([]int, 0, len(saiyans))
    for _, saiyan := range saiyans {
        powers = append(powers, saiyan.Power)
    }
    return powers
}
```

切片作为一个数组的封装是一个非常有用的概念。很多语言都有类似的概念。javascript和ruby的数组都有一个 `slice` 方法。你也可以在ruby中通过 `[START..END]` 得到一个切片，或者在python中通过 `[START:END]` 得到一个切片。然而，在一些语言中，切片确实是从原始数组拷贝而来的新数组。如果我们使用 `ruby` ，下面代码将输出什么？

```
scores = [1,2,3,4,5]
slice = scores[2..4]
slice[0] = 999
puts scores
```

答案是 `[1, 2, 3, 4, 5]` 。因为切片 `slice` 是由值拷贝组成的一个全新数组。现在，同等情况下看go：

```
scores := []int{1,2,3,4,5}
slice := scores[2:4]
slice[0] = 999
fmt.Println(scores)
```

输出是 `[1, 2, 999, 4, 5]` 。

这会如何改变你的代码。例如，很多函数需要一个位置参数。在javascript中，如你我们想在前五个字符后查找一个字符串中的第一个空白符（对，切片也当字符串处理），我们可以这样写：

```
haystack = "the spice must flow";
console.log(haystack.indexOf(" ", 5));
```

在go中，我们使用切片：

`strings.Index(haystack[5:], " ")`

从上面例子中，我们可以看出 `[X:]` 表示 `X` 到结尾的一种缩写。而 `[:X]` 是开始到 `X` 的一种缩写。不像其他语言，go不支持负值索引。如果我们想要切片所有元素，但除了最后一个，我们可以这样写：

```
scores := []int{1, 2, 3, 4, 5}
scores = scores[:len(scores)-1]
```

上述是一种从一个乱序的切片中去除一个值的有效方法。

```
func main() {
    scores := []int{1, 2, 3, 4, 5}
    scores = removeAtIndex(scores, 2)
    fmt.Println(scores)
}

func removeAtIndex(source []int, index int) []int {
    lastIndex := len(source) - 1
    //swap the last value and the value we want to remove
    source[index], source[lastIndex] = source[lastIndex], source[index]
    return source[:lastIndex]
}
```

最后，我们已经学习了切片，我们来学习一下另外一个常用的内置函数：`copy`。`copy` 是众多函数中重点显示出切片如何改变我们代码方式的函数之一。正常情况下，拷贝一个数组到另外一个数组的方法需要5个参数：`source`，`sourceStart`，`count`，`destination` 和 `destinationStart`。但是在切片中，我们只需要2个参数：

```
import (
    "fmt"
    "math/rand"
    "sort"
)

func main() {
    scores := make([]int, 100)
    for i := 0; i < 100; i++ {
        scores[i] = int(rand.Int31n(1000))
    }
    sort.Ints(scores)
    worst := make([]int, 5)
    copy(worst, scores[:5])
    fmt.Println(worst)
}
```

花点时间研究上面的代码。试着改变一些代码。如果你使用`copy(worst[2:4], scores[:5])`方式去复制看看会发生什么，或者试着复制多于或者少于5个值到 `worst` 。

链接

- 目录
- 上一节：数组
- 下一节：映射

3.3 映射

go语言中的映射在其他语言中叫字典或哈希表。正如你想象的：你可以定义一个键和值，然后可以从映射中获取、设置和删除这个值。

和切片一样，映射也是可以通过 `make` 创建。让我们看看下面这个例子：

```
func main() {
    lookup := make(map[string]int)
    lookup["goku"] = 9001
    power, exists := lookup["vegeta"]
    // 打印：0和false
    // 0代表一个整数型的默认值
    fmt.Println(power, exists)
}
```

使用 `len` 可以获得映射中键的个数。使用 `delete` 可以删除映射中的一个键值对。

```
// 返回 1
total := len(lookup)
// 没有返回值, 可以调用一个不存在的键
delete(lookup, "goku")
```

映射是动态增长的。然后，我们也可以在使用 `make` 时传递第二个参数设置映射的初始大小：

`lookup := make(map[string]int, 100)`

如果你能知道你的映射有多少个键，定义时指定一个初始大小可以获得一定的性能提升。

当你希望将一个映射作为一个结构体的字段时，你可以这样定义：

```
type Saiyan struct {
    Name    string
    Friends map[string]*Saiyan
}
```

初始化上面定义的结构体的一种方式：

```
goku := &Saiyan{
    Name:    "Goku",
    Friends: make(map[string]*Saiyan),
}
goku.Friends["krillin"] = ... //可以创建Krillin
```

这里还有提供了另外一种方式去定义并初始化一个映射。类似 `make` ，这种方式是针对映射和数组。我们可以声明一个复合文字：

```
lookup := map[string]int{
    "goku":  9001,
    "gohan": 2044,
}
```

在 `for` 循环中，使用 `range` 关键字也可以遍历一个映射：

```
for key, value := range lookup {
    ...
}
```

需要注意的是，编译映射并不是有序的。每次遍历映射时，返回的键值对都是随机的顺序。

链接

- 目录
- 上一节：切片
- 下一节：指针类型和值类型

3.4 指针类型和值类型

在第二章我们已经学习了何时传递和赋值一个指针类型或者值类型。现在我们在数组和映射方面再谈论该话题。我们该使用哪一个？

```
a := make([]Saiyan, 10)
//或者
b := make([]*Saiyan, 10)
```

很多开发人员认为传递 `b` 至一个函数或者让函数返回 `b` 效率更高。然而，这里传递或者返回的都是一个切片的拷贝，这本身就是一个引用。所以就传递或者返回这个切片而言，没有什么区别。

当你改变一个切片或者映射的值时，你会看见不同。在这点上，同样的逻辑，我们在第二章看到已经适用。所以是否定义一个数组指针还是一个数组值主要归结于如何使用单个值，而不是你如何使用数组或者映射本身。

链接

- 目录
- 上一节：映射
- 下一节：继续之前

3.5 继续之前

数组和映射在go语言中的工作方式和其他语言很类似。如果你使用过动态数组，可能需要一点点适应，但是`append`应该能解决你大多数不适。如果我们抛开数组表面的语法，我们就会发现切片。切片是相当强大的，使用切片对你代码的整洁性有着非常巨大的影响。

这里有一些边界例子我们没有涉及到，但是你不太可能遇见这些例子。另外，如果你遇到了，希望我们已经打下的基础能让你理解这是怎么回事。

链接

- 目录
- 上一节：指针类型和值类型
- 下一章：代码组织和接口

4.0 代码组织和接口

现在是时候来看看如何组织我们的代码了。

链接

- 目录
- 上一章：继续之前
- 下一节：包

4.1 包

为了理解更复杂的库和组织系统，我们需要学习包。在go语言中，包名和你的go语言工作空间的目录结构有关。如果我们想要构建一个购物系统，我们开始可能以 `shopping` 作为包名，并将源代码放入 `$GOPATH/src/shopping` 。

我们不想把一切东西都放入这个文件夹中。例如，可能我想在它自己的文件夹隔离一些数据库逻辑。要达到此目的，我们可以在 `$GOPATH/src/shopping` 中创建一个子目录 `db` 。在这个子目录中的文件的包名可以简单的称为 `db` 。但是如果其他的包想要引用这个包，需要包含 `shopping` 包，我们必须这样导入 `shopping/db` 。

换句话说，当你命名一个包时，通过使用关键字 `package` ，你只需要提供单个值，而不是一个完整的层次结构（例如："shopping"或者"db"）。但是当你导入一个包时，你需要指定一个全路径。

让我们试试，在你的go工作空间的 `src` 目录（我们在开始介绍），创建一个新的文件夹 `shopping` ，并在 `shopping` 中创建一个目录 `db` 。

在 `shopping/db` 中创建一个文件 `db.go` ，并写入下面的代码：

```
package db

type Item struct {
    Price float64
}

func LoadItem(id int) *Item {
    return &Item{
        Price: 9.001,
    }
}
```

需要注意包名和文件夹名是相同的。而且很明显我们实际并没有连接数据库。这里使用这个例子只是为了展示如何组织代码。

现在，在 `shopping` 文件中创建一个 `pricecheck.go` 文件，并写入下面的代码：

```
package shopping

import (
    "shopping/db"
)

func PriceCheck(itemId int) (float64, bool) {
    item := db.LoadItem(itemId)
    if item == nil {
        return 0, false
    }
    return item.Price, true
}
```

大多数人容易认为，导入 `shopping/db` 有点特殊，因为我们已经在 `shopping` 文件夹里面了。实际上，我们是导入 `$GOPATH/src/shopping/db` ，这就意味如果你的工作空间 `src/test` 文件下有一个 `db` 包，这也可以通过 `test/db` 导入这个 `db` 包。

如果你打算构建一个包，你只需要做以上的步骤。要构建一个可执行文件，你还得需要一个 `main` 。我最喜欢的是在 `shopping` 目录创建一个子目录 `main` ，并创建一个 `main.go` 文件，写入以下代码：

```
package main

import (
    "fmt"
    "shopping"
)

func main() {
    fmt.Println(shopping.PriceCheck(4343))
}
```

现在你可以运行进入你的 `shopping` 项目，输入以下命令运行你的代码：

`go run main/main.go`

4.1.1 循环导入

当你开始写更复杂的系统时，你一定会遇到循环导入。当 `A` 包导入 `B` 包， `B` 包又导入 `A` 包时就会发生这种情况（通过其他包直接或者间接引起）。这种情况编译器不允许。

让我们改变我们的 `shopping` 结构体引起这种错误。

将 `Item` 的定义从 `shopping/db/db.go` 移到 `shopping/pricecheck.go` 。你的 `pricecheck.go` 文件如下：

```
package shopping

import (
    "shopping/db"
)

type Item struct {
    Price float64
}

func PriceCheck(itemId int) (float64, bool) {
    item := db.LoadItem(itemId)
    if item == nil {
        return 0, false
    }
    return item.Price, true
}
```

如果你试着去运行代码，你将会从 `db/db.go` 得到一个关于 `Item` 未定义的错误。这是有意义的。`db` 包不再存在 `Item` ；它已经被移动到 `shopping` 包。我们需要去改变 `shopping/db/db.go` 为：

```go
package db

import (
    "shopping"
)

func LoadItem(id int) *shopping.Item {
    return &shopping.Item{
        Price: 9.001,
    }
}
```

现在再运行一下代码，你会得到一个严重错误： `import cycle not allowed` ，不允许循环导入。要解决这个问题，需要导入另外一个包，这个包定义了共享结构体。你的目录结构应该像下面这样：

```
$GOPATH/src
    - shopping
        pricecheck.go
        - db
            db.go
        - models
            item.go
        - main
            main.go
```

`pricecheck.go` 仍然导入 `shopping/db` ，但是 `db.go` 现在通过导入 `shopping/models` 替换之前的 `shopping` ，这样就可以消除循环导入。由于我们将共享结构体 `Item` 移到了 `shopping/models/item.go` ，我们需要改变 `shopping/db/db.go` ，让它从 `models` 包引用 `Item` 结构体。

```go
package db

import (
    "shopping/models"
)

func LoadItem(id int) *models.Item {
    return &models.Item{
        Price: 9.001,
    }
}
```

你平时共享的模块不仅仅是 `models` ，所以你可以还有其他类似的文件夹如 `utilities` 之类。关于这些共享包的一个重要规则就是：他们不应该从 `shopping` 或者子包中导入任何东西。在一些小节，我们会看到通过接口可以帮助我们清理这些类型的依赖关系。

4.1.2 可见性

go语言使用了一种简单的规则来规定类型或者函数是否对外部的包可见。如果你命名类型或者函数时以一个大写字母开头，那么这个类型和函数就是可见的。如果使用一个小写字母开头，那么就是不可见的。

结构体的字段也使用相同的方式。如果一个结构体的字段名是以小写字母开头的，那么只有在同一个包中的代码才能访问这些字段。

例如，在我们的 `items.go` 文件中有一个函数类似这样：

```
func NewItem() *Item {
    // ...
}
```

可以通过 `models.NewItem()` 调用这个函数。但是如果这个函数名为 `newItem` ，那么我们从其他的包是不能调用这个函数的。

继续改变 `shopping` 包中函数名、类型名和字段名。例如，如果你将结构体 `Item` 的 `Price` 字段改成 `price` ，你会得到一个错误。

4.1.3 包管理

我们已经使用过go语言的命令如 `go run` 和 `go build` ，go还有一个子命令 `get` 可以用来获取第三方库的代码。 `go get` 支持很多种协议，但是这个例子中，我们将通过它从github上得到一个库，这意味着你必须在你的电脑上安装git。

加入你已经安装了git，在命令行中输入：

`go get github.com/mattn/go-sqlite3`

`go get` 将得到这些远程文件并将它们保存在你的工作空间。现在去查看你的 `$GOPATH/src` 。除了我们已经创建的 `shopping` 工程，你会看见一个 `github.com` 文件夹。明确在文件夹中你会看见一个 `mattn` 文件夹，它包含了一个 `go-sqlite3` 文件夹。

我们已经学习了如何导入一个包到我们的工作空间。现在如果想使用我们刚刚获取的 `go-sqlite3` 包，可以通过以下方式导入：

```
import (
    "github.com/mattn/go-sqlite3"
)
```

我知道这看起来像是一个网址，但它是实际存在的，当go编译器在 `$GOPATH/src/github.com/mattn/go-sqlite3` 目录中能发现这个包时，你可以很简单的导入 `go-sqlite3` 包。

4.1.4 依赖管理

`go get` 有一些其他的花样。如果我们在一个项目中执行 `go get` ，它会扫描所有文件并查找所有导入的第三方库，然后下载这些第三方库。某种程度上说，我们自己的源代码变成一个 `Gemfile` 或者 `package.json` 。

执行 `go get -u` 将更新你的包（或者你可以通过 `go get -u FULL_PACKAGE_NAME` 更新指定的包）。

最后，你可能发现了 `go get` 的一些不足。首先，它不能指定一个修订，它会一直指向 `master/head/trunk/default` 。这是一个严重的问题，尤其当你有2个项目需要同一个库的不同版本时。

为了解决这个问题，你可以使用一个第三方的依赖管理工具。虽然还不太成熟，但是有2个依赖管理工具比较有前景，即goop和godep。更完整的列表可以参考go-wiki。

链接

- 目录
- 上一节：代码组织和接口
- 下一节：接口

4.2 接口

接口是一种类型，它只定义了声明，没有具体实现。例如：

```
type Logger interface {
    Log(message string)
}
```

你也许觉得奇怪，怎样做有什么用。接口可以是你的代码从具体的实现中去耦。例如，我们可以会有很多种类型的 `loggers` ：

```
type SqlLogger struct { ... }
type ConsoleLogger struct { ... }
type FileLogger struct { ... }
```

如果在编程时使用接口，而不是它们具体的实现，我们可以很容易的改变和测试我们代码，但是对我们的代码没有任何影响。

你会如何使用？就像其他类型一样，它可以作为一个结构体的字段：

```
type Server struct {
    logger Logger
}
```

或者一个函数参数（或者返回值）：

```
func process(logger Logger) {
    logger.Log("hello!")
}
```

在c#或者java中，当一个类实现了一个接口时，必须明确定义出：

```
public class ConsoleLogger : Logger {
    public void Logger(message string) {
        Console.WriteLine(message)
    }
}
```

这也会促进小和集中的接口，go的标准库充满了接口。 `io` 包又一些受欢迎的例如 `io.Reader` 、 `io.Writer` 和 `io.Closer` 。如果你在写一个函数，函数参数只会调用 `Close` 。你完全可以传递一个 `io.Closer` 接口类型，而不管你使用的具体类型。

接口也可以组合。也就是说接口可以有其他接口组成。例如， `io.ReadCloser` 就是由接口 `io.Reader` 和 `io.Closer` 接口组成。

最后，接口常用于避免循环导入。由于接口没有具体的实现，所以他们的依赖性有限。

- 目录
- 上一节：包
- 下一节：继续之前

4.3 继续之前

最后，当你试着用go写一些简单的项目之后，你会习惯在go语言的工作空间中组织代码的方式。最重要是的记住go语言中的包名和你的目录结构有密切关系（不仅仅在一个项目中，在整个工作空间都如此）。

go语言处理类型的可见性方法是简单有效的。也是一致的。还有一些内容我们没有介绍，例如常量和全局变量，但是不用担心，它们的可见性也是遵循同样的规则。

最后，如果你不熟悉go语言中的接口，你可能需要花一些时间去感受它们。无论如何，当你首次看见一个函数例如 `io.Reader` 之类，你会发现你自己感激作者不苛求超过他或她需要的。

链接

- 目录
- 上一节：接口
- 下一章：go花絮

5.0 go花絮

在这章，我们会介绍一些go语言的特性，这些特性不适合使用在其他地方。

链接

- 目录
- 上一章：继续之前
- 下一节：错误处理

5.1 错误处理

go语言没有异常处理，一般通过返回值处理错误。例如 `strconv.Atoi` 函数将一个字符串转换成一个整数：

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

func main() {
    if len(os.Args) != 2 {
        os.Exit(1)
    }
    n, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println("not a valid number")
    } else {
        fmt.Println(n)
    }
}
```

你也可以自己创建一个错误类型；唯一的要求就是它必须实现内置类型 `error` 接口：

```
type error interface {
    Error() string
}
```

通常情况下，我们可以通过导入 `errors包` ，然后使用包中的 `New` 函数创建一个自己的错误类型：

```
import (
    "errors"
)

func process(count int) error {
    if count < 1 {
        return errors.New("Invalid count")
    }
    ...
    return nil
}
```

go的标准库就是通过这种模式使用错误类型变量。例如，在 `io` 包中，有一个 `EOF` 变量定义如下：

```
var EOF = errors.New("EOF")
```

这是一个包级别变量（它是定义在函数外面），该变量是可以访问的（因为第一个字母是大写字母）。当我们从文件或者标准输入中读取数据时，很多函数都能返回这种错误。如果有上下文关系，你也应该使用这个错误。作为消费者，我们可以使用这个单例：

```
package main

import (
    "fmt"
    "io"
)

func main() {
    var input int
    _, err := fmt.Scan(&input)
    if err == io.EOF {
        fmt.Println("no more input!")
    }
}
```

最后要指出的是，go语言有 `panic` 和 `recover` 函数。 `panic` 类似抛出异常，而 `recover` 类似捕获异常；但是很少使用它们。

链接

- 目录
- 上一节：go花絮
- 下一节：defer

5.2 defer

尽管go语言提供了垃圾回收器，但是有些资源需要我们明确的去释放。例如，当我们处理完一个文件事，需要使用 `Close()` 关闭文件。这类代码总是危险的。首先，当我们写一个函数时，很容易忘记关闭在第十行声明的某些东西。另外，一个函数可能会有多个返回点。go的解决方法会是提供了 `defer` 关键字：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Open("a_file_to_read")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()
    // 读这个文件
}
```

如果你试着运行上面的代码，你可能会得到一个错误（因为文件不存在）。关键在于展示 `defer` 是如何工作。无论如何你的 `defer` 都会在方法返回时得到执行，虽然这有点极端。但是这可以帮你在初始化的附近释放资源，并且可以实现多个返回点。

链接

- 目录
- 上一节：错误处理
- 下一节：go语言风格

5.3 go语言风格

大多数使用go语言开发的程序都遵循相同的格式化规则，也就是说，使用 `tab` 缩进并且花括号和语句在同一行。

我知道，你有属于自己的风格，并且你也想坚持下去。我曾经很长一段时间也是这样的。但是我很高兴我最终还是屈服了。就是因为 `go fmt` 命令，这个命令易用且权威（所以没有人会为了毫无意义的偏好而争论）。

当你在工程内部，你可以通过下面的命令将工程下所有文件使用相同的格式化规则：

```
go fmt ./...
```

试一试，这个命令会给你代码添加缩进，自动对齐你的声明语句并将包导入按字母顺序排序。

链接

- 目录
- 上一节：defer
- 下一节：初始化的if

5.4 初始化的if

go支持一种稍有不同的 `if` 语句，一个值可以在条件语句执行前定义并初始化：

```
if x := 10; count > x {
    ...
}
```

这是一种很愚蠢的例子，多数情况下，你会这样做：

```
if err := process(); err != nil {
    return err
}
```

比较有趣的是， `if` 语句中定义并初始化的值在 `if` 语句之外是不可用的，但是可以在 `else if` 和 `else` 语句中使用。

链接

- 目录
- 上一节：go语言风格
- 下一节：空接口和转换

5.5 空接口和转换

在大多数面向对象语言中，都有一种内置的基类，叫 `object` ，它是所有其他类的超类。但是go语言不支持继承，所以没有类似超类的概念。go拥有一个没有任何方法的空接口： `interface{}` 。因为每种类型都实现了空接口的0个方法，并且接口都是隐式实现，所以每种类型都实现了空接口的条约。

如果我们愿意，我们可以通过下面声明方式写一个 `add` 函数：

```
func add(a interface{}, b interface{}) interface{} {
    ...
}
```

将一个空接口变量转换成一个指定的类型，你可以使用 `.(TYPE)` :

```
return a.(int) + b.(int)
```

你也可以通过switch使用强大的类型转换：

```
switch a.(type) {
    case int:
        fmt.Printf("a is now an int and equals %d\n", a)
    case bool, string:
        // ...
    default:
        // ...
}
```

你慢慢会发现，空接口的使用会超出你的预期。不可否认，这样会让代码看起来不整洁。反复转换一个值不好看并且容易出错，但有时候在静态类型语言中，这是唯一的选择。

链接

- 目录
- 上一节：初始化的if
- 下一节：字符串和字节数组

5.6 字符串和字节数组

字符串和字节数组有密切关系，我们可以轻易的将它们转换成对方：

```
stra := "the spice must flow"
byts := []byte(stra)
strb := string(byts)
```

事实上，这也是大多数类型的转换方式。一些函数明确指定一个 `int32` 或者 `int64` 或者相应的无符号类型。你可能会发现你自己不得不像下面一样：

```
int64(count)
```

尽管如此，当提到字节数组和字符串时，这可能是你会一直接触的东西。在你使用 `[]byte(X)` 或者 `string(X)` 时务必注意，你创建的是数据的拷贝。这是由于字符串的不可变性。

当字符串有由 `unicode` 字符码 `runes` 组成时。如果你计算字符串的长度时，可能得到的结果和你期待的不同。下面结果是输出3：

```
fmt.Println(len(""))
```

如果你通过 `range` 遍历一个字符串，你将得到 `runes` ，而不是 `bytes` 。当然，当你将一个字符串转换成一个 `[]byte` 时，你将得到正确的数据。

链接

- 目录
- 上一节：空接口和转换
- 下一节：函数类型

5.7 函数类型

函数是一流的类型：

```
type Add func(a int, b int) int
```

可以使用在任何地方-比如结构体字段、参数或者一个返回值。

```
package main

import (
    "fmt"
)

type Add func(a int, b int) int

func main() {
    fmt.Println(process(func(a int, b int) int {
        return a + b
    }))
}
func process(adder Add) int {
    return adder(1, 2)
}
```

像这样使用函数可以使你在一些特定实现时减少代码的耦合性，就像使用接口实现那样。

链接

- 目录
- 上一节：字符串和字节数组
- 下一节：继续之前

5.8 继续之前

我们已经学习了go编程的很多内容。显而易见，我们看见了错误处理的行为和资源释放如链接或者打开文件。很多人不喜欢go语言的错误处理方式。它让人觉得这是一种退步。有些时候，我同意这种说法。然而，我也发现这会导致代码更易读。 `defer` 是一种不常见但很实用的资源管理手段。事实上，它不仅仅可以进行资源管理。你可以使用 `defer` 完成任何目的，例如当一个函数退出时打印日志记录。

当然，我们还没有学习go提供的所有花絮。但是无论你遇到什么你应该可以轻松应对。

链接

- 目录
- 上一节：函数类型
- 下一章：并发

6.0 并发

go语言常被描述成一种适用于并发的语言。主要是因为go语言在2种强大的机制上提供了简单的语法支持：go协程和通道。

链接

- 目录
- 上一章：继续之前
- 下一节：go协程

6.1 go协程

go协程类似一个线程，但是go协程是由go自己调度，而不是系统。在协程中的代码可以和其他代码并发执行。让我们看一个例子：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("start")
    go process()
    time.Sleep(time.Millisecond * 10) // this is bad, don't do this!
    fmt.Println("done")
}

func process() {
    fmt.Println("processing")
}
```

这个例子有一些有趣的事，但是最重要的是了解我们是如何启动一个go协程。我们只是简单的将 `go` 关键字附在我们想要执行的函数前面即可。如果我们只想执行一小段代码，例如上面的例子一样，我们可以使用一个匿名函数。需要注意的是，匿名函数不只是在go协程中使用，其他地方也可以。

```
go func() {
    fmt.Println("processing")
}()
```

go协程很容易创建且开销较小。最终多个go协程将会在同一个底层的系统线程上运行。这也常称之为 `M:N` 线程模型，因为我们有 `M` 个应用线程（go协程）运行在 `N` 个系统线程上。结果就是，一个go协程的开销和系统线程比起来相对很低（一般都是几KB）。在现代的硬件上，有可能拥有成千上万个go协程。

另外，这里还隐藏了映射和调度的复杂性。我们只需要说这段代码需要并发执行，然后让go自己去处理。

如果我们回到刚刚的例子中，你将会注意到我们使用了 `Sleep` 让程序等待了几毫秒。这是因为主进程在退出前协程才有机会去执行（主进程在退出前不会等待所有协程都执行完毕）。为了解决这个问题，我们必须让代码协同。

链接

- 目录

- 上一节：并发
- 下一节：同步

6.2 同步

创建一个协程没有什么难度，并且启动很多协程开销也不大。但是，并发执行的代码需要协同。为了帮助我们解决这个问题，go提供了通道（channels）。在学习通道之前，我认为有必要先学习了并发编程的基本知识。

在编写并发执行的代码时，你需要特别的关注在哪里和如何读写一个值。出于某些原因，例如没有垃圾回收的语言，需要你从一个新的角度去考虑你的数据，总是警惕着可能存在的危险。例如：

```go
package main

import (
    "fmt"
    "time"
)

var counter = 0

func main() {
    for i := 0; i < 2; i++ {
        go incr()
    }
    time.Sleep(time.Millisecond * 10)
}

func incr() {
    counter++
    fmt.Println(counter)
}
```

你认为会输出什么？

如果你觉得输出是 `1` 和 `2` ，不能说你对或者错。如果你运行上面的代码，确实如此。你很有可能得到那样的输出。但是，实际上这个输出是不确定的。为什么？因为我们可能有多个（这个例子中是2个）go协程同时写同一个变量 `counter` 。或者更糟的情况是一个协程正在读 `counter` ，而另一个协程正在写 `counter` 。

这确实危险吗？绝对是的。 `counter++` 似乎看起来只是一行简单的代码，但是实际上它被拆分为很多汇编指令，具体依赖于你运行的软件和硬件平台。在上面的例子中，确实在大多数情况下运行良好。然而，另外一个可能的结果是 `counter` 等于0 时被2个协程同时读取，那么你将得到一个输出是 `1,1` 。还有更坏的结果，例如系统崩溃或者得到一个任意值然后自增。

在并发程序中，如果想安全的操作一个变量，唯一的手段就是读取该变量。你可以有任意多的程序去读，但是写必须是同步的。这里有几种方式实现，包括使用依赖于特殊cpu架构的一些真正的原子操作。然而，大多数时候都是使用一个互斥锁：

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var (
    counter = 0
    lock    sync.Mutex
)

func main() {
    for i := 0; i < 2; i++ {
        go incr()
    }
    time.Sleep(time.Millisecond * 10)
}
func incr() {
    lock.Lock()
    defer lock.Unlock()
    counter++
    fmt.Println(counter)
}
```

互斥锁可以使你按顺序访问代码。因为 `sync.Mutex` 默认值是没有锁的，所以我们简单的定义了一个锁 `lock sync.Mutex` 。

看起来似乎很简单？上面的例子带有欺骗性。当做并发编程时会发现一些列很严重的bug。首先，那些代码需要被保护一直都不是容易发现。虽然它可能是想使用一个低级锁（这个锁涉及了很多代码），这些潜在出错的地方是我们做并发编程首先要去考虑的。我们常常想要精确的锁，或者我们最终由一个10车道的高速突然转变成一个单车道道路。

另外一个问题是如何处理死锁。当使用一个锁时，这没有问题，但是如果你在代码中使用2个或者更多的锁，很容易出现一种危险的情况，即协程A拥有锁 `lockA` ，想去访问锁 `lockB` ，同时协程B拥有 `lockB` 并需要访问锁 `lockA` 。

实际上使用一个锁也有可能发生死锁问题，即当我们忘记释放它时。但是这和多个锁引起的死锁为比起来，危害性不大（因为这真的很难发现），但是当你试着运行下面代码时，你可以看见发生了什么：

```
package main

import (
    "sync"
    "time"
)

var (
    lock sync.Mutex
)

func main() {
    go func() { lock.Lock() }()
    time.Sleep(time.Millisecond * 10)
    lock.Lock()
}
```

迄今为止有很多并发编程我们都还没用见过。首先，由于我们可以同时有多个读操作，有一种常见的锁叫读写锁。它主要提供2中锁功能：一个锁定读和一个锁定写。在go语言中，`sync.RWMutex` 就是这种锁。另外 `sync.Mutex` 结构不但提供了 `Lock` 和 `Unlock` 方法，也提供了 `RLock` 和 `RLock` 方法，这里的 `R` 代表 `Read` 。虽然读写锁很常用，但是他们也给开发者带来一些额外的负担：我们不但要关注我们正在访问的数据，而且也要关注如何访问。

此外，部分并发编程不只是通过为数不多代码按顺序的访问变量，也需要协调多个go协程。例如，休眠10毫秒不是一种优雅的方法。如果一个go协程消耗的时间不止10毫秒呢？如果go协程消耗少于10毫秒，我们只是浪费了cpu？又或者可以等待go协程运行完毕，我们告诉另外一个go协程：嗨，我有一些新数据给你处理？

所有的这些事在没有通道（channels）的情况下都是可以实现的。当然，对于更简单的例子，我认为你应该使用基本的功能例如 `sync.Mutex` 和 `sync.RWMutex` 。但是在下一节我们将看到，通道的主要目的是为了使并发编程更简洁和不易出错。

链接

- 目录
- 上一节：go协程
- 下一节：通道

6.3 通道

并发编程的挑战主要在于数据分享。如果你的go协程没有共享数据，你就不需要担心同步他们。但是，对于所有的系统，这不是一个选择。实际上，考虑到很多系统以完全相反的目标构建：通过多个请求来分享数据。一个内存中的缓存或者一个数据库都是这方面的好例子。这正变得越来越流行的事实。

通道通过使共享数据不相干让并发编程变得更加理智。一个通道是一个通信管道，它用于go协程之间传递数据。换句话说，一个go协程可以通过通道传递数据给另外一个go协程。其结果就是，在任何时候，仅有一个go协程可以访问数据。

通道所有其他的东西一样，也有类型。这种类型类型的数据就是将要在我们通道中传递的数据。例如，创建一个通道，这个通道可以用来传递一个整数，我们可以这样：

```
c := make(chan int)
```

这个通道的类型是 `chan int` 。因此，将这个通道传递给一个函数是，可以这样声明：

```
func worker(c chan int) { ... }
```

通道支持2中操作：接收和发送。我们可以使用下面方式往通道发送数据：

```
CHANNEL <- DATA
```

可以使用下面方式从通道接收数据：

```
VAR := <-CHANNEL
```

箭头的方向就是数据的流动方向。当发送数据时，数据流入通道。当发送数据时，数据是流出通道。

最后，在查看我们的第一个例子之前，我们需要知道从一个通道接收或者发送数据时会阻塞。当我们从一个通道接收数据时，直到数据可用go协程才会继续执行。类似的，往一个通道发送数据时，在数据被接收之前go协程也不会继续执行。

考虑到洗个系统的输入数据，我们想通过不同的协程去处理这些数据。这是一种常见的需求。如果通过go协程接收输入的数据并进行数据密集型处理。那么在客户端会有超时风险，首先，我们将写出我们的 `worker` 。这可以是一个简单的函数，但是我会让它变成一个结构体的部分，因为我们之前从来没有这样使用过go协程：

```
type Worker struct {
    id int
}

func (w Worker) process(c chan int) {
    for {
        data := <-c
        fmt.Printf("worker %d got %d\n", w.id, data)
    }
}
```

我们的 `worker` 很简单。它会一直等待数据，直到数据可用然后处理它。它在一个循环中，永远尽职的等待更多的数据并处理。

为了使用上面的 `worker` ，我们首先要做的是启动一些 `worker` ：

```
c := make(chan int)
for i := 0; i < 4; i++ {
    worker := Worker{id: i}
    go worker.process(c)
}
```

然后我们可以给它们一些工作：

```
for {
    c <- rand.Int()
    time.Sleep(time.Millisecond * 50)
}
```

这是完整的代码，运行它：

```go
package main

import (
	"fmt"
	"math/rand"
	"time"
)

func main() {
	c := make(chan int)
	for i := 0; i < 5; i++ {
		worker := &Worker{id: i}
		go worker.process(c)
	}

	for {
		c <- rand.Int()
		time.Sleep(time.Millisecond * 50)
	}
}

type Worker struct {
	id int
}

func (w *Worker) process(c chan int) {
	for {
		data := <-c
		fmt.Printf("worker %d got %d\n", w.id, data)
	}
}
```

我们不知道哪个 `worker` 将获得数据。我们所知道的是，go语言确保了往一个通道发送数据时，仅有一个单独的接收器可以接受。

需要指出的是通道是共享状态的唯一方式，通过通道我们可以并发安全的发送和接收数据。通道提供了我们需要的所有同步代码，并且也确保在任意的特定时刻只有一个go协程可以访问一个特定的数据。

6.3.1 带缓存的通道

在上面的代码中，如果输入的数据超过我们可以处理的数据会发生什么？你可以模拟这种场景，在 `worker` 接收到数据后，让 `worker` 执行 `time.Sleep` ：

```go
for {
	data := <-c
	fmt.Printf("worker %d got %d\n", w.id, data)
	time.Sleep(time.Millisecond * 500)
}
```

在 `main` 函数中会发什么呢？接收用户的输入数据（这里通过一个随机的数字生成器模拟）会被阻塞，因为往通道发送数据时没有可用的接收者。

在这种情况下，你需要确保数据被处理，你可能想要让客户端阻塞。在其他情况下，你可能愿意不确保数据被处理。这里有一些流行的策略能完成此事。首先是将数据缓存起来。如果没有 `worker` 可用，我们想将数据暂时存放在一个有序的队列中。通道拥有这种内置额度缓存能力。当我们使用 `make` 创建一个通道时，我们可以指定通道的长度：

```
c := make(chan int, 100)
```

你可以改变，但是你将注意到处理过程仍然震荡。当缓存的通道不能添加更多的容量；它只不过是为挂起的作业提供一个队列，以一种更好的方式处理数据突然飙升。在我们示例中，我们可以连续不断的发送更多的数据，并且 `worker` 也能处理这些数据。

虽然如此，事实上，我们可以查看通道的长度了解到带缓存的通道的缓冲作用：

```
for {
    c <- rand.Int()
    fmt.Println(len(c))
    time.Sleep(time.Millisecond * 50)
}
```

你可以看到带缓存的通道长度在不断增大，直到装满为止，此时，往通道发送的数据又开始被阻塞。

6.3.2 select

即使借助缓存，有一点需要指出的是，我们需要开始丢弃一些消息，我们不能使用一个无限大的内存，并指望人工的释放它。所以我们使用go语言的 `select` 。

在语法结构上， `select` 看起来有点类似 `switch` 。通过 `select` ，我们能写出一些针对通道不可写情况下的代码。首先，让我们去掉我们通道的缓存，这样可以更清晰的看到 `select` 是如何工作的。

```
c := make(chan int)
```

接下来，我们修改 `for` 循环：

```
for {
    select {
        case c <- rand.Int():
            //可选的代码
        default:
            //这里可以留下空行以丢弃数据
            fmt.Println("dropped")
    }
    time.Sleep(time.Millisecond * 50)
}
```

我们每秒往通道中发送20个信息，但是我们的程序每秒只能处理10个信息；因此，有一半的信息被丢弃。

这仅仅只是我们使用 `select` 完成一些事的开始。使用 `select` 的最主要目的是通过它管理多个通道。给定多个通道， `select` 将阻塞直到有一个通道可用。如果没有可用的通道，当提供了 `default` 语句时，执行该分支。当多个通道都可用时，选择其中的一个通道是随机的。

很难想出一个简单的例子来证明这种行为，因为这是一种高级特性。在下一小节可能有助于说明这个问题。

6.3.3 超时

我们已经学习了带缓存的信息并且也简单的丢弃它们。另外一种比较流行的做法是使用超时。我们将阻塞一段时间，但是不是一直阻塞。这在go中很容易实现。老实说，这个语法有点难于接受，但是它是比较灵活和有用的特性，我基本不能没有它。

为了是阻塞达到最大值，我们可以使用 `time.After` 函数。让我们看看它会发生什么神奇的事。通过使用这种方式，我们发送器变成：

```
for {
    select {
    case c <- rand.Int():
    case <-time.After(time.Millisecond * 100):
        fmt.Println("timed out")
    }
    time.Sleep(time.Millisecond * 50)
}
```

`time.After` 将返回一个通道，所以我们可以对它使用 `select` 语句。这个通道在经过指定的时间后会被写入。就是这样。没有什么比这个更神奇了。如果你依然觉得奇怪，这里实现了一个 `after` ，如下所示：

```
func after(d time.Duration) chan bool {
    c := make(chan bool)
    go func() {
        time.Sleep(d)
        c <- true
    }()
    return c
}
```

回到我们的 `select` 语句，这里有一些好玩的东西。首先，如果你在后面添加 `default` 分之会发生什么？你能猜到吗？试试。如果你不确定会发生什么，记住如果通道不可用的话， `default` 分支会被立即执行。

此外， `time.After` 是一个 `chan time.Time` 类型的通道。在上面的例子中，我们将发送给通道的值简单丢弃掉。如果你想，你也可以获取到这个值：

```
    case t := <-time.After(time.Millisecond * 100):
        fmt.Println("timed out at", t)
```

密切注意我们的 `select` 。注意我们正在往 `c` 中发送数据，但是从 `time.After` 收取。不管我们是从通道中接收数据、发送数据或者收发数据， `select` 工作机制都一样：

- 第一个可用的通道被选中
- 如果多个通道可用，随机选中一个通道
- 如果没有通道可用， `default` 分之被执行
- 如果没有 `default` 分之， `select` 将阻塞

最后，在 `for` 循环中使用 `select` 也是比较常见的，例如：

```
for {
    select {
    case data := <-c:
        fmt.Printf("worker %d got %d\n", w.id, data)
    case <-time.After(time.Millisecond * 10):
        fmt.Println("Break time")
        time.Sleep(time.Second)
    }
}
```

链接

- 目录
- 上一节：同步
- 下一节：继续之前

6.4 继续之前

如果你才开始进入并发编程的世界，它看起来似乎势不可挡。它绝对需要非常多的关注。`go`目标就在于让并发更容易。

go协程很有效的抽象了我们需要并发执行的代码。通道通过消除共享数据，帮助我们消除了一些当数据共享时导致的严重bug。这不仅仅是消除bug，但是它改变了我们如何进行并发编程。你可以认为是通过信息传递实现并发编程，而不是那些容易出错的代码。

话虽如此，我仍然在广泛使用`sync`和`sync/atomic`包中的同步原语。我觉得比较重要的是通过使用这2中方式比较舒适。我支持你首先关注通道，但是当你遇到一些需要短暂的锁的简单例子时，你也可以考虑下使用互斥锁或者读写锁。

链接

- 目录
- 上一节：通道
- 下一章：结论

6.4 总结

我最近听go被描述成一门无聊的语言。无聊时因为很容易学，容易写，更重要的是容易读。也许，我实际帮了个倒忙。我已经用了3章谈论类型并介绍了如何声明变量。

如果你有静态类型语言的编程背景，最好情况下，我们看到的大多数可能只是复习一下而已。go语言让指针用起来更明显且易用了，并且go将数组封装成切片，对经验丰富的java和c#程序员来说，不是什么压倒性的优势。

如果你主要是使用动态型语言，你可能觉得有点不一样。这是一个公平的学习。不仅仅是各种各样的语法声明和初始化。尽管作为一个go语言的粉丝，尽管作为Go的粉丝，我发现，对于所有的简单性的进展，也有一些不太简单。虽然如此，这里也涉及到一些基本规则（比如你可以使用:=声明变量，但是只能声明一次）和基本理解（比如使用new(x)或者&X{}只能分配内存，但是切片、映射和通道需要更多的初始化所以使用make）。

除此之外，go语言让我们以一种简单有效的方式组织代码。接口，基于返回值的错误处理方式，通过defer管理资源，并且以一种简单的方式实现组合。

最后但是也最重要的是go内置支持并发。关于go协程没有什么要说的了，除了协程简单有效（无论如何使用简单）。这是一个很好的抽象。通道更为复杂。我一直认为在使用高水平封装之前先理解最基本使用方法。我认为不通过通道学习并发编程是很有用的。但是，对我来说，我觉得通道的实现方式不像一个简单的抽象。它们几乎都是自己的基本构建块。我这样说是因为它们改变了你如何编写和思考并发编程。考虑到并发编程是多么的不易，这肯定是一件好事。

链接

- 目录
- 上一章：继续之前